VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Bakalářská práce

Rok 2010

Zbyněk Složil

VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra Informatiky

Server pro protokol XmPP s podporou multimédií

Xmpp Protocol Server with Support of Multimedia

Rok 2010 Zbyněk Složil

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Zbyněk Složil**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Server pro protokol XmPP s podporou multimédií
XmPP Protocol Server with Support of Multimedia

Zásady pro vypracování:

Navrhněte a pokuste se realizovat server pro operační systém Linux, který by tvořil základ pro komunikaci klientů prostřednictvím protokolu XmPP známého u klientů Jabber. Pokuste se také implementovat funkci, která by umožňovala nejen přenos textových zpráv, ale i multimediálních dat. Funkce koncipujte tak, ať jsou kompatibilní s klientskými aplikacemi, která vytváří v rámci své práce Zdeněk Parma (PAR114).

1. Popište protokol XmPP.
2. Navrhněte základní funkce serveru.
3. Navrhněte rozšíření o možnost přenosu multimediálních dat.
4. Pokuste se navržené funkce realizovat.
5. Dbejte na dobrou dokumentaci.

Seznam doporučené odborné literatury:

Dle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Seidl**

Datum zadání: 20.11.2009
Datum odevzdání: 07.05.2010

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: Zbyněk Složil

Rád bych tímto poděkoval Ing. Davidu Seidlovi za odborné vedení mé bakalářské práce a trpělivost, kterou se mi laskavě věnoval. Také děkuji kolegům Zdeňkovi Parmovi a Tomašovi Sikorovi za spolupráci.

Abstrakt

Práce se věnuje seznámení s protokolem XMPP a má za cíl dosáhnout implementace XMPP serveru běžícím na Linuxu. Server poběží jako konzolová aplikace a poskytne připojeným klientům přeposláním zpráv.

Další část práce je věnována návrhu implementace multimediální části, která by umožňovala audio/video konektivitu mezi klienty.

Klíčová slova

XMPP, XML, iksemel, XMPP server, socket, pthread

Abstract

This work introduces with XMPP protocol and its goal is to achieve implementation of XMPP server running on Linux. Server will run as console application and it will provide message routing to connected clients.

Next part is dedicated to concept of multimedial part , witch will provide audio/video conectivity between clients.

Keywords

XMPP, XML, iksemel, XMPP server, socket, pthread

Seznam použitých symbolů a zkratek

.NET – „dotnet“, platforma pro vývoj aplikací

AES – Advanced Encryption Standard, standard pro šifrování

DOM – Document Object Model, objektově orientovaná reprezentace XML

GNU GPL – GNU General Public License, všeobecná veřejná licence GNU

GNU LGPL – GNU Lesser General Public License, všeobecná veřejná licence GNU

ICQ – „I Seek You“, software pro instant messaging

IM – Instant Messaging, internetová služba pro komunikace mezi klienty

IP – Internet Protocol, protokol pro přenos po síti

JID – Jabber ID, jméno určující uživatele v síti Jabber

MD5 – Message-Diggest 5, hašovací funkce pro tvorbu „otisků“ ze vstupních dat

MSNP – Microsoft Notification Protocol, protokol pro komunikaci

POSIX – Portable Operating System Interface, přenositelné rozhraní pro operační systémy

RTP – Real Time Protocol, protokol pro posílání audio a video dat po internetu

SASL – Simple Authentication and Security Layer, ověřovací metoda mezi klientem a serverem

SAX – Simple API for XML- způsob procházení XML dokumentem

TCP – Transmission Control Protocol, potvrzované posílání paketů

TLS – Transport Layer Security, kryptografický protokol

UDP – User Datagram Protocol, nepotvrzené posílání paketů

UIN – Unified Identification Number, jedinečný identifikátor uživatele v síti ICQ

UTF-8 – UCS Transformation Format, způsob kódování řetězců znaků

W3C – World Wide Web Consortium, mezinárodní konsorcium pro vývoj webových standardů

XML – Extensible Markup Language, značkovací jazyk

XMPP – Extensible Messaging and Presence Protocol**,** protokol pro komunikaci

Obsah

1. Úvod

Internetové služby jsou jedním ze základů dnešní společnosti. Obrovské množství lidí denně prochází webové stránky, prahnouc po nových informacích, nakupují v elektronických obchodech, v nichž je vše, na co člověk dokáže pomyslet, ale hlavně komunikuje s ostatními. Velké množství sítí, které se rozrůstají do každého koutu naší planety, sblížili lidi tak, že i obyčejný člověk v České republice může klidně mluvit s jiným na druhé polokouli. Komunikace je pilířem společnosti a internet je prostředek, který to velmi zjednodušuje. Dnes dokonce nejde jen o posílání zpráv nebo „chatování“ jak jej všichni známe, ale i o přenos hlasu nebo videa. Písmo nevyjádří tón v hlase nebo gestikulaci volajících. Technika pokročila tak, že velké množství lidí vlastní v dnešní době vysokorychlostní linky a většina kořenových vedení internetu dosahuje požadované propustnosti, které zvládají obrovské množství přenosu emailů, souborů, hlasových i video konferencí po celém světě. Proto nic nebrání tomu, aby se vytvořila softwarová vrstva, která by tyto možnosti využila.

Jak už jsem psal komunikace je jedna z mnoha činností, kterou lidé na internetu využívají. Mnoho z nich aktivně používá například webová fóra, chatovací místnosti, IM[1] klienti nebo dnes tolik rozšířené sociální sítě. Různé druhy těchto komunikačních technologii se vyvíjely postupně nebo vznikly spojením už existujících druhů. Webová fóra jsou místnosti, ve kterých se řeší problémy nebo jen tak diskutuje. Fórum je řešeno jako dlouhodobá diskuze a zprávy napsané do těchto fór je možné zobrazit i mnoho let po napsání. Další druhem jsou chatovací místnosti, které se soustřeďují pro posílání zpráv ve skupině přihlášených lidí v reálném čase. Pro mě důležitým a často používaným druhem komunikace po internetu je IM klient, který vlastní snad každý uživatel připojený k internetu. Posledním typem je sociální síť, dnes tak hojně využívaná. Její výhodo[1]u je celkové obecné řešení. Uživatel má vše po ruce a má mnoho způsobu předávání informací s ostatními – zprávy, statusy, vlastní charaktery, přidávání do skupin, sdílení audia/videa aj.

Mě, jak už jsem psal, zajímají IM klienti a komunikace mezi nimi. Tato oblast je rozsáhlá a existuje obrovské množství způsobů řešení tohoto problému. Na internetu je velké množství těchto klientů a protokolů, které poskytuji a řídí přenosy mezi samotnými IM klienty nebo servery. Na ukázku uvedu u nás nejznámější a mnou vybraný představím nakonec.

OSCAR – je protokol, který využívá hlavně IM klient ICQ (I seek you). Oba produkty, jak protokol tak klient, je chráněn komerční licencí a nedovoluje jiným vývojářům nahlédnout do kódu. ICQ klient získal u nás velkou oblibu stejně jako v Rusku nebo v Německu. Klient se přihlašuje pomocí svého UIN[2] a přidává si do svého seznamu kontaktů jiné UIN identifikujíc své přátele. UIN je číslo, které se jako jediné v profilu nedá změnit. Při řešení architektury je jako vždy použit model klient-server, kde klientem jsou připojení uživatelé a serverem je pár počítačů společnosti vlastnící licenci. Jak lze vypozorovat, architektura je centralizovaná a při výpadku padá celá síť. Přenos není vůbec šifrovaný, což dneska představuje velkou hrozbu a míra bezpečnosti autentizace také dosahuje minima. Sama společnost zakazuje používání ve firmách, proto jedinými zákazníky jsou domácí uživatelé. Ještě je důležité dodat, že například zprávy, které se přepošlou, se ihned stávají vlastnictvím společnosti. Z těchto důvodu klienta zásadně nedoporučuji, už jen proto, že se potýká s krádežemi účtů a spamem. Oficiálním klientem je pouze ten od vlastníka licence, ale existují i projekty, které řeší propojení s touto sítí.

MSNP – protokol, celým názvem Microsoft Notification Protocol, který se stal jedním z nejpoužívanějších na světě. Vyvíjí ho společnost Microsoft společně se svým klientem

1 Instant messaging, internetová služba pro komunikace mezi klienty.
2 Unified Identification Number, unikátní identifikace uživatele.

Windows Live Messenger pomocí své technologii .NET[3]. U nás nevešel tolik v oblibu jako ICQ, ale řady uživatelů se stále rozšiřují. Existuje mnoho jiných IM klientů, kteří tento protokol spolehlivě používají, ale naplno ho pouze zmíněný oficiální klient. Pro něj není problém sdílet nástěnku s kresbami, soubory, audio/video komunikace nebo se synchronizovat s programem Outlook[4].

Skype – IM klient, kterého uvádím jako příklad u nás často používaného multimediálního komunikátoru na síti. Tak jako ICQ i protokol programu Skype je uzavřený a nikdo neví co se přes něj přesně odesílá. Podobnost s ICQ má i v centralizovanosti. Centrální server ovšem oproti ICQ slouží většinou pouze k autentizaci uživatele a spojení se provádí přímo mezi klienty, nebo přes jiného klienta, který má veřejnou adresu IP. V tom se také nachází další problém, pokud chce mít klient spolehlivé přímé spojení s jiným klientem, musí mít veřejnou IP adresu a hrozí mu že se stane uzlem pro jiné potřebné klienty s neveřejnou IP. Přesto je Skype často používán a jeho přenosy patří k bezpečnějším (šifrování pomocí AES[5]).

XMPP[1] - mezi lidmi známější jako síť Jabber. Je to vlastně jen popsaný standard, podle kterého se řídí přenosy mezi serverem a klientem nebo mezi dvěma servery. To přináší velké množství implementací a proto kvalita a bezpečnost je zaručena pouze pokud je správně implementován server a používáte správný klient. Na internetu je ovšem mnoho produktů, takže výběr je široký a každý si vybere dle svých potřeb. U nás nebyl nikdy tolik rozšířen jako ostatní protokoly a používala ho jen úzká skupinka lidí. Dnes se stal tento standard nejpoužívanějším na světě. Sociální síť Facebook na něj přešla a díky tomu se dá k této sociální síti připojit i pomocí oblíbeného IM klienta. Velká výhoda je také v identifikaci uživatele. Ten je v systému zaregistrován jako „klient@doména_serveru“ a na rozdíl od čísla se to lépe pamatuje. Podobnost na emailovou adresu je zřejmá, proto se často provádí na serveru propojení s poštovním serverem. Tou zásadní výhodou je decentralizovanost celého systému. Pokud se zhroutí jeden server, ostatní běží dál a výpadek nemusí ani zaznamenat.

Jak lze z posledního odstavcem vypozorovat, XMPP je pouze standard a nikomu nezakazuje vytvářet si své vlastní implementace serveru nebo klientů. Proto jsem si i já se svými kolegy Zdeňkem Parmou a Tomášem Sikorou vybral za bakalářskou práci projekt implementace XMPP klienta a serveru. Rozvrhli jsme si a přerozdělili práci podle svých zájmů a znalostí. Představím cíl každé práce a její základní obsah pro celkový pohled na projekt.

Moje úloha

Moje práce v projektu je implementace XMPP serveru, jehož cílem je poskytnutí základních komunikačních dat klientovi a přeposílání zpráv mezi nimi po síti. Jako rozšíření by měla být multimediální komunikace. Server byl programován především pro práci s klientem od mých kolegů. Při implementaci nebyla použita skoro žádná pomocná knihovna pro usnadnění práce, a proto musel být server vytvořen od úplných základů, to jak část komunikace, tak i řešení samotných XML[6] dokumentů. Zatím chybí odesílání chybových zpráv zpět ke klientovi, což i když patří k důležité části protokolu, neomezuje to funkčnost. Za to server poskytuje autentizaci klienta, stažení seznamu kontaktů, presenci a odesílání samotných zpráv. Je to teprve první verze mé implementace, která se stala kostrou většího projektu.

3 „dotnet“, platforma pro vývoj aplikací.
4 Poštovní klient vyvíjený firmou Microsoft.
5 Advanced Encryption Standard, standard pro šifrování s až 256 bitovým klíčem.
6 Extensible Markup Language, značkovací jazyk, používaný pro serializaci dat.

Práce navazuje těsně s prací Zdeňka Parmy a jeho jádrem klienta XMPP. Mezi námi jsou přeposílány XML zprávy, které můj server rozkóduje a přepošle na požadované místo. Tím je komunikace mezi námi přesně daná a nevznikají žádné rozpory.

Úloha Zdeňka Parmy (citace)

Jádro XMPP klienta tvoří spojení se s libovolným XMPP serverem. Správou kontaktů získaných od serveru. Posílání zpráv a výměna multimediálních dat. Součástí práce bylo, seznámit se s XMPP protokolem. Vybrat vhodnou knihovnu pro realizaci jádra klienta a pokusit se jí implementovat. Dohodnout se s kolegou Tomášem Síkorou na komunikaci mezi uživatelským rozhraním a jádrem klienta. Vymyslet strategii, která bude použita při přenosu multimediálních dat a pokusit se jí realizovat. Vytvořit testovací program pro ověření implementovaných funkcí.

Výsledné jádro XMPP klienta podporuje standardy XMPP Core a XMPP IM. Je schopen se spojit s libovolným XMPP serverem podporujícím tyto standardy. Jádro klienta je plně propojeno s uživatelský rozhraním a samostatně funguje pouze v testovacím programu. Strategie pro přenos multimediálních dat byla vytvořena. Tato strategie se zakládá na využití real time protokolu (RTP), ale nakonec se nepodařilo jí implementovat. Jádro klienta bylo testováno na několika veřejných XMPP serverech. Nejdůkladnější testování probíhalo se serverem kolegy Zbyňka Složila.

Úloha Tomáše Sikory (citace)

GUI klienta je řešeno pomocí jazyka QT. QT se hodilo pro tuto práci zdaleka nejvíce a pro popis důvodů a volby samotné, se podívejte prosím do mé práce. Řešení je multiplatformní (s nutností rekompilace a linkování příslušné knihovny, jak pro Microsoft Windows tak pro Linux) a je zde jedna verze běžící v konzoli. Jako contactlist jsem implementoval komponentu používanou klientem PSI a využívám, některých jeho funkcí. Díky vlastnostem QT se mi podařilo splnit prakticky vše předsevzaté, ale bohužel jsem neimplementoval multimediální funkce a nedodělal jsem konzolové rozhraní tak jak bych chtěl (například v podobě nCurses (obojí se bohužel nestihlo dodělat do prezentovatelné formy nicméně jsem rozebral případné řešení alespoň teoreticky). Klient je připojitelný na jakýkoli server podporující jabber (alespoň teoreticky, povedlo se mi připojit na všechny, které jsem testoval) a většinu transportních serverů. Stylování klienta je řešeno pomocí přiloženého .qss souboru a editace probíhá na základě vlastností specifikovaných tvůrcem jazyka.

Má práce je úzce spjata s projektem Zdeňka Parmy. Mám za úkol poskytovat grafický výstup aplikace a využít funkcí jeho jádra klienta.

2. Návrh řešení

2.1. Volba jazyka a operačního systému

Obecnou funkcí mé aplikace je komunikační server, který se stará o to, aby data, které přijdou, byla vyhodnocena a poté poslána zpět danému klientovi. Toto přeposílání probíhá pomocí XML souborů. Naskýtá se tedy mnoho způsobů implementace a to jak ve výběru programovacího jazyka tak operačního systému, na kterém by běžel. S výběrem operačního systému poté souvisí i způsob spojení po sítí (internetu) či paralelizace programu.

Existuje mnoho programovacích jazyků, ale v mém případě měly dva navrch, Java a C++. Java, i když středně složitý jazyk mnohdy s vyřešenými algoritmy a datovými kontejnery, se zdál velmi nevhodný pro implementaci serveru jakožto aplikace běžící na virtuálním stroji Javy, kde by zbytečně ztrácela na výkonu, což u serveru není přípustné. Další výhodou by asi bylo jednoduché multiplatformní použití, což u tohoto projektu není prioritou. Naproti C++ je jazyk, který velmi dobře znám a vyhovoval výkoností. Těžší je ovšem najít správné knihovny, které by řešili mnohé důležité problémy.

Dalším výběrem byl operační systém. I když Microsoft Windows mi je dobře znám, zvolil jsem implementaci na Linuxu. Důvodem byly hlavně znalosti o způsobu implementace na tomto operační systému. Během studia se mnohé části řešené v mé aplikaci učily i v předmětech, a proto jsem s nimi byl dobře seznámen. Jde hlavně o paralelizaci a komunikaci po síti pomocí socketů. I když je pravda že Windows implementuje tyto části podobně ne-li stejně.

Po vyřešení těchto dvou otázek jsem se vrhl na problémy návrhu samotné implementace.

2.2. Různé koncepce řešení

Dalším důležitým krokem bylo, jak vlastně celou aplikaci rozdělit tak aby byla tvořena z částí, které by usnadnily vývoj a přehlednost. Koncept základního problému je velmi obecný (Obr.1).

Obr.1 Základní koncept

2.2.1. První koncept

Jeden ze způsobu je vytvoření aplikace pouze s jedním procesem. Celý program by běžel ve velkém cyklu a na určitém místě čekal na jakoukoli událost z venčí. K tomu by se použila funkce select nebo poll, který by kontroloval množinu socketů. Vždy by se pustila příslušná část kódu a poté znova najela na čekající funkci. Tento způsob má mnoho nevýhod. V dnešní době je jedno-jádrových procesorů velmi poskrovnu (hlavně na serverech), proto by aplikace nikdy nevyužila celý výkon (vytěžovala by asi pouze jedno jádro). Dále by byla brzděna pomalostí I/O operacemi na síti. To samé, pokud by aplikace zpracovávala nějakou obtížnou úlohu (parserovara složité XML nebo vytvářela pro něj data), nemohla by během této činnosti přijímat a vysílat zprávy. Aplikace by musela mít obrovské systémové buffery, aby toto zpomalování ošetřila. Proto jsem tento koncept ihned zavrhl.

2.2.2. Druhý koncept

Z minulého konceptu tedy vyplývá jasné řešení. Oddělením I/O části od zbytku aplikace bych dal možnost, aby jádro programu pracovalo nezávisle. Přidáním bufferu mezi tyto části bych umožnil

vzájemnou komunikaci a také bych tím ulevil systémovým I/O bufferům. Důsledkem by také bylo zvýšení výkonu na vice-jádrových (více-procesorových) strojích. Bohužel v koncepci se skrývá jedna chyba a to v samotné implementaci funkce select nebo poll. Ty totiž pracují pouze s identifikátorem souboru (file-descriptor) a po bližším prozkoumání nahrnuté I/O oblastí jsem narazil na problém. V naslouchací části není problém, naslouchaný socket se přidá k množině socketů. To samé už připojené klientské sockety ze kterých se čte. Problém však nastává při zápisu, při kterém se nečeká na změnu na socketu, ale na změnu v bufferu.

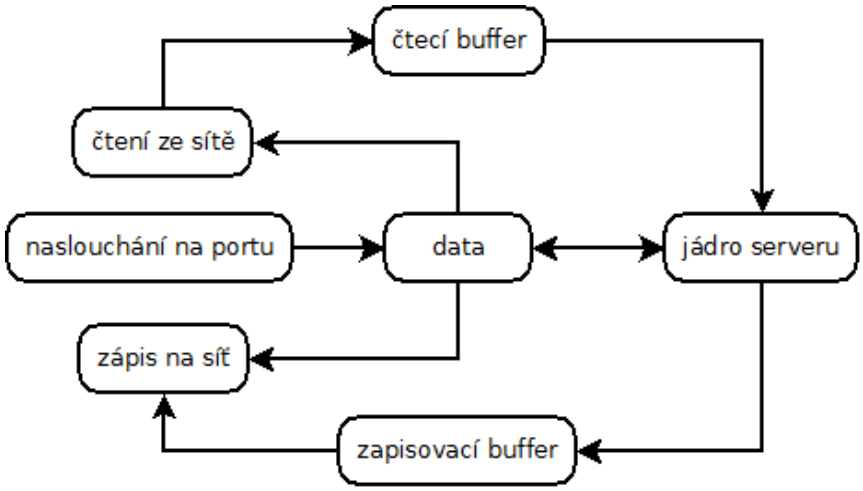
S tím také přišla volba způsobu paralelizace. Měl jsem na výběr vlákna nebo tvorbu procesů. Pokud bych zvolil vytvoření dvou procesů, mohl bych jako buffery použít roury, které jsou v UNIX systému definované jako identifikátor souboru. Potom bych mohl výstup z roury přidat do funkce select (poll) a zápis by čekal na změnu v tomto identifikátoru. Mě se však paralelizace pomocí procesů a bufferů jako rour moc nezamlouvala, proto jsem koncept ještě trochu upravil pro použití s vlákny.

2.2.3. Třetí koncept

V minulých konceptech se počítalo s tím, že existuje množina socketů všech klientů co se připojí k serveru. Pomocí funkce select (poll) se příchozí spojení roztřídí a poté se čtená data pošlou na vyřešení jádru. Dalším způsobem je pro každé příchozí spojení vytvořit vlákno, které by řešilo daného klienta. Tento způsob má ovšem velké množství nevýhod. Na všech systémech je omezený počet běžících vláken, proto by se u většího počtu připojení mohlo dojít k hranici. Dále by každé vlákno muselo také být jádrem (podobně jako první koncept). Pokud bych zvolil jedno vlákno jako jádro, nastal by podobný problém se zapisováním jako ve druhém konceptu. Ten by se dal vyřešit vytvoření dvou vláken pro každého klienta, jedno pro zápis, druhé čtení a naslouchání (tím se by se o polovinu snížilo maximum možných spojení). Naskytne se ale problém velkého množství bufferů, které by řešili komunikaci mezi vlákny. Řešení pomocí tvorby procesů místo vláken by už vůbec nepřipadalo v úvahu, veškerý výkon serveru by byl zaplněn tvorbou a rušením velkých datových oblastí těchto procesů. Celý tento koncept je tedy špatný a nelze využít.

2.2.4. Konečný koncept

Jako nejlepší a nejzajímavější řešení jsem zvolil trochu upravený druhý koncept, ve kterém jsem rozdělil čtení a zápis dat a navíc přidal další vlákno pro naslouchání na síti (pro pozdější multimediální naslouchání). Komunikaci mezi vlákny zařizují dva buffery, které vyrovnávají rychlostní rozdíly mezi vlákny. Vazby jsou zobrazeny na Obr.2.



Obr.2 Konečný koncept

3. Technologie a jejich výběr

3.1. Protokol XMPP

3.1.1. Obecně o protokolu

The Extensible Messaging and Presence Protocol (XMPP)[4] je otevřený protokol pro komunikaci v reálném čase. Mezi klienty a servery se předávají XML soubory a na rozdíl od jiných protokolů je XMPP decentralizovaný a při výpadků neovlivní ostatní servery. Je to vlastně obrovská síť serverů se svými klienty a tak jako poštovní servery se pouze dorozumívají pomocí domén. Vyvinula ho open-source komunita pod názvem Jabber, ze kterého protokol XMPP vyšel. Přesněji řečeno, XMPP by se dal nazvat standardem, na kterém běží síť Jabber.

Jak už jsem psal v úvodu, každý uživatel má své uživatelské jméno, které se skládá z vlastního nicku (unikátní na serveru) a domény serveru, na který se připojuje. Tomu se říká „bare JID“. Výhodou XMPP je, že i klientská aplikace se dokáže identifikovat. Po JID se přidá název zdroje, který například definuje, která aplikace nebo systém se připojil. Tomu se říká „full JID“. Přináší mnoho výhod jako připojení vícekrát na jeden účet. Příklad JID:

klient_jedna@server.cz - bare JID
klient_dva@server.cz/miranda - full JID

Protokol nemusí sloužit jen ke komunikaci mezi lidmi, využívá se i ke komunikaci mezi hrami nebo k použiti automatických služeb. Je dobré také poznamenat že celý XMPP je kódován do UTF-8, proto české znaky nejsou žádným problémem.

Seznam výhod a nevýhod XMPP protokolu:

Výhody:

- decentralizovanost – Jak už jsem zmínil, síť se skládá z mnoha serverů, které mají své vlastní klienty. Pokud dojde k poškození jednoho z nich, neohrozí to ostatní. Podobně jako poštovní servery, XMPP spolu komunikují pomocí domén. Tato architektura také přispívá k otevřenosti systému, takže každý může spustit svůj vlastní server a komunikovat s ostatními.
- bezpečnost přenos – Přenos po síti může být šifrovaný. Každý server nabídne své možné způsoby šifrování (např. TLS) a poté klient vybírá jeden z nich.
- autentizace – Kódované heslo například do MD5.
- ověřena architektura – Celý systém je velmi dobře odzkoušen a mnoho známých produktu přešlo na tento standard.
- otevřenost – Systém je naprosto otevřený, každý může nahlédnout do specifikací protokolu. Také mnoho implementaci je vytvářeno jako open-source projekty.
- zvláštní služby – Existuje velké množství rozšíření, které protokolu XMPP přidává nové vymoženosti (transporty, RSS čtečky, …).
- možnost přispět – Každý může vymyslet další rozšíření protokolu a zveřejnit ho.

Nevýhody:

- obecná neznalost – Celá síť Jabber byla odjakživa používána jenom úzkou komunitou, i když poslední dobou to už nebývá pravda.
- klient – Neexistuje žádný klient, který by využil všechny schopnosti tohoto protokolu a přinutil

nového uživatele připojit se k této síti.

- heartbeat – Velká nevýhoda této sítě. Celý systém je postaven na posílání zpráv mezi klienty (přeposlané serverem). Každý klient se při odpojení musí odhlásit posláním zprávy a tím dá vědět ostatním, že se stane nedostupným. Pokud je však vypnutí nečekané a daná zpráva se nestihne poslat, pro ostatní uživatele je klient stále připojen. Toto řeší technika heartbeat, která zkouší zda je klient stále dostupný, a pokud je náhle odpojen, funkce to detekuje. XMPP jde však směrem minimalizace posílaných dat, což heartbeat porušuje.
- JID – I když se zdá že decentralizovanost má hlavně přínosy, jedna nevýhoda zde je. JID uživatele obsahuje i doménu serveru, na kterém se účet spravuje, proto pokud chce uživatel přejít na jiný, třeba že má lepší služby, musí změnit své JID.

XMPP má mnoho implementací a většinou se dodávají jako plug-iny do různých komunikačních programků (Psi, QIP, Miranda). Velkou výhodou jsou také transporty, které vytváří portování na straně serveru s jinými protokoly.

3.1.2. Jazyk XML

Dříve než podrobněji popíšu standard XMPP, bylo by dobré seznámit se s XML dokumentem na kterém je založeno. Je to značkovací jazyk vyvinut konsorciem W3C[7]. Dnes je velmi často používán pro přenosy a ukládání souborů jakožto univerzální formát. Jeho velkou výhodou je že schéma si tvoří každý sám, ale pokud je XML validní tak jde kdykoli rozluštit.

Pro mě víc než XML bylo důležité jeho rozkouskování (parserování), a proto se zaměřím na tuto část. Existují dva hlavní způsoby rozdělení dokumentu XML:

DOM vznikl za pomocí W3C a je to oficiální způsob rozdělování a tvorby XML dokumentů. Algoritmus pracuje tak, že vytvoří v paměti obraz XML dokumentu ve tvaru stromu. Procházení tímto stromem je poté velmi pohodlné.

Výhody: - přehledné zanořování elementů
- může přidávat a odebírat elementy ze stromu, proto je DOM jediný způsob jak jednoduše vytvořit XML dokument

Nevýhody: - může zabírat velký prostor v paměti
- při tvorbě stromu může být aplikace pomalejší

SAX [5] není oficiálním standardem W3C a ani se nesnaží DOM nahradit. Jeho použití je jiné a pouze dává další způsob jak XML rozdělit. SAX neukládá XML do stromu, pouze prochází dokument a zasílá event (např. volá funkce) podle toho, jaký typ dat se řeší. Například pokud SAX nalezne nějaký tag, zavolá funkci pro řešení tagů, která zjistí zda byl párový nebo nepárový. Podobně projde celý dokument.

Výhody: - netvoří se strom, proto zabírá méně místa v paměti (velmi výhodné u velkých dokumentů)
- vysoká rychlost algoritmu
- možnost zaměřit se jen na část dokumentu

Nevýhody : - nelze dodatečně upravovat dokument
- nedovolí vrátit se už k vyřešeným částem

Ve svém programu používám oba způsoby. Nevýhoda u implementace algoritmu DOM v knihovně iksemel je, že nedokáže rozdělit XML zprávu, která není celistvá (validní). Protože první

7 World Wide Web Consortium, mezinárodní konsorcium, které pomáhá vyvíjet webové standardy

zpráva která se přijímá od klienta je jen první část párového tagu (zbylá je poslána při ukončení spojení), DOM se zasekne a s ním i celé vlákno. Z toho důvodu pro určení prvotního tagu používám SAX, který zavolá funkce řešící daný problém. V těchto funkcích se už DOM používá, ale pouze u zpráv které mají validní části.

3.1.3. Bližší prozkoumání protokolu

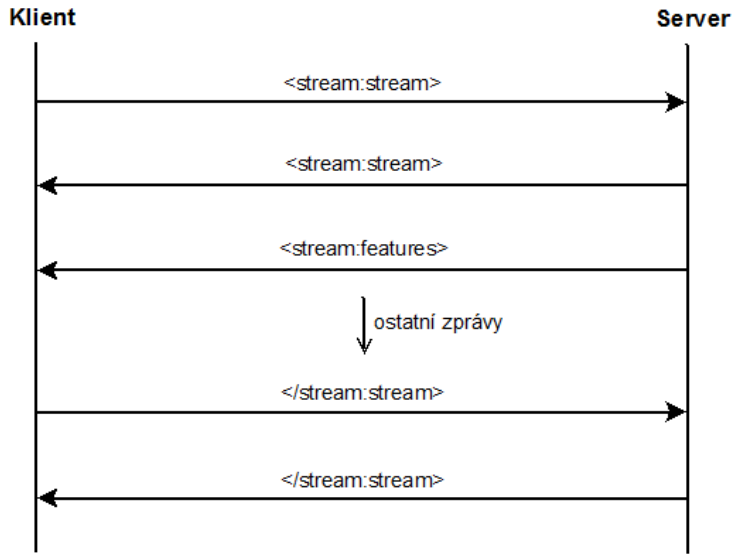
XMPP protokol je rozděleny na RTF dokumenty, což jsou normy, které namísto mezinárodních norem (např. ISO) vytvářejí obyčejní lidé a nezaručují, že je bude každý používat, pouze poskytují návrh. Základním RTF dokumentem u XMPP je RTF 3920 a pro samotný IM je RTF 3921. Budu zde spíš řešit svůj způsob implementace, protože protokol je velmi obsáhlý a já použil jen nutné části.

RTF 3920 [6]

Základní norma, která definuje spojení klienta se serverem a šifrovaná spojení. Pokud se bude chtít klient spojit už zmíněným šifrovaným spojením, musí se serverem domluvit a spojení znova navázat v šifrované podobě. Tato část ovšem nebyla v mé implementaci prioritou. Navíc je v ní obsažen způsob autorizace k serveru.

Navázání a ukončení spojení (Obr.3)

Základním tagem pro zahájení a ukončení spojení je stream.



Obr.3 Stream

Příklad průběhu:

Server přijme:

```
<?xml version='1.0' ?>
<stream:stream to='lizols.cz' xmlns='jabber:client'
xmlns:stream='http://etherx.jabber.org/streams'  xml:lang='en' version='1.0'>
```

Zpráva poslána při navázání spojení klienta se serverem. Server porovná zda je zpráva pro něj (atribut to) a pošle zpět odpověď že přijal žádost:

```
<stream:stream  xmlns:stream='http://etherx.jabber.org/streams'  id='A'  xmlns='jabber:client'
from='lizols.cz' version='1.0'>
```

Zpráva poslána zpět ke klientovi. Server k ní přidá atribut id pro určení streamu na klientské straně. Navíc odešle nabídku druhů služeb.

```
<stream:features>
        <auth xmlns='http://jabber.org/features/iq-auth'/>
</stream:features>
```

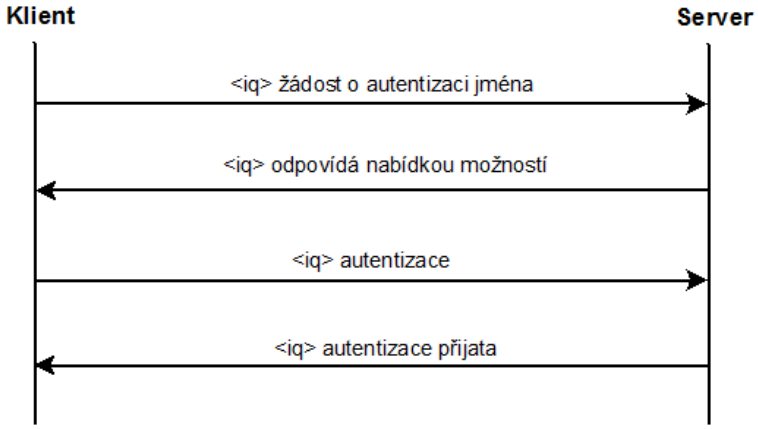
V mém případě nabízím pouze službu autorizace. Nyní se přeposílají ostatní XML zprávy. Pokud chce klient spojení ukončit musí poslat:

```
</stream:stream>
```

Server jako odpověď pošle to samé zpět. Tím se komunikace ukončí a server uzavírá nepotřebný socket pro další použití.

Autentizace (Obr.4)

Autentizace se provádí pomocí iq zpráv.



Obr.4 Autentizace

Příklad průběhu:

Server přijme:

```
<iq to='lizols.cz' id='uid:4bdf5228:6b8b4567' type='get' xmlns='jabber:client'>
        <query xmlns='jabber:iq:auth'>
                <username>parma</username>
        </query>
</iq>
```

Zpráva poslaná jako žádost o autentizaci uživatele. Zde se také zkontroluje zda je uživatel na disku vůbec uložen. Pokud ano, odpovídá server nabídkou možnosti autentizace:

```
<iq id='uid:4bdf5228:6b8b4567' type='result' xmlns='jabber:client'>
        <query xmlns='jabber:iq:auth'>
                <username>parma</username>
                <password/>
                <resource/>
        </query>
</iq>
```

V mém případě nabízím pouze poslání nezašifrovaného hesla (další možnost je třeba pomocí MD5[8]) a

8 Message-Diggest 5, hašovací funkce pro tvorbu „otisků“ ze vstupních dat (například pro kontrolní součty).

také žádám o název zdroje (resource). Klient odpovídá:

```
<iq id='uid:4bdf5228:327b23c6' type='set' xmlns='jabber:client'>
        <query xmlns='jabber:iq:auth'>
                <username>parma</username>
                <password>zdenek</password>
                <resource>gloox</resource>
        </query>
</iq>
```

Zpráva obsahuje vyplněné hodnoty, o které server žádal. Pokud by server nabízel více způsobu poslání hesla, klient si vybere jeden z nich. Pokud heslo souhlasí, server pošle zpět odpověď:

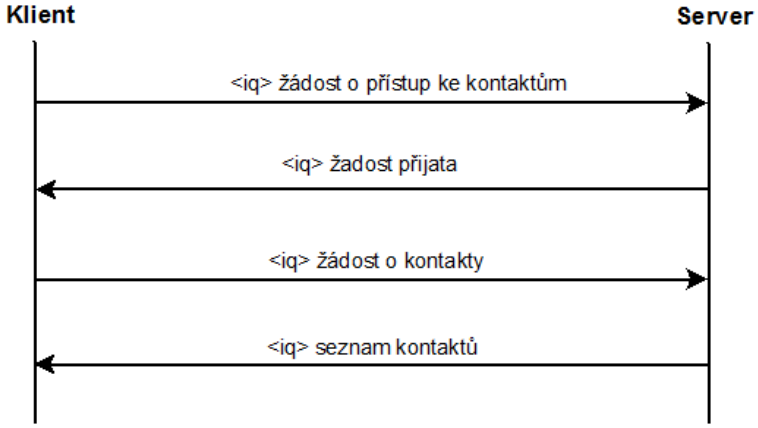
```
<iq id='uid:4bdf5228:327b23c6' type='result' xmlns='jabber:client'/>
```

RTF 3921 [7]

Jedna z dalších norem, která přidává definici kontaktů (roster), presence (presence) a posílání zpráv (message).

Získaní kontaktů (Obr.5)

Tak jako autentizace i kontakty se získávají pomocí iq zpráv.



Obr.5 Kontakty

Příklad průběhu:

Server přijme:

```
<iq id='uid:4bdf5228:643c9869' type='get' from='parma@lizols.cz' xmlns='jabber:client'>
        <query xmlns='jabber:iq:private'>
                <roster xmlns='roster:delimiter'/>
        </query>
</iq>
```

Klient zde žádá o přístup ke svým kontaktům. Server pozitivně odpovídá:

```
<iq id='uid:4bdf5228:643c9869' type='result' xmlns='jabber:client' from='parma@lizols.cz'>
        <query xmlns='jabber:iq:private'>
                <roster xmlns='roster:delimiter'/>
        </query>
</iq>
```

Server přijme:

```
<iq id='uid:4bdf5228:66334873' type='get' from='parma@lizols.cz' xmlns='jabber:client'>
        <query xmlns='jabber:iq:roster'/>
</iq>
```

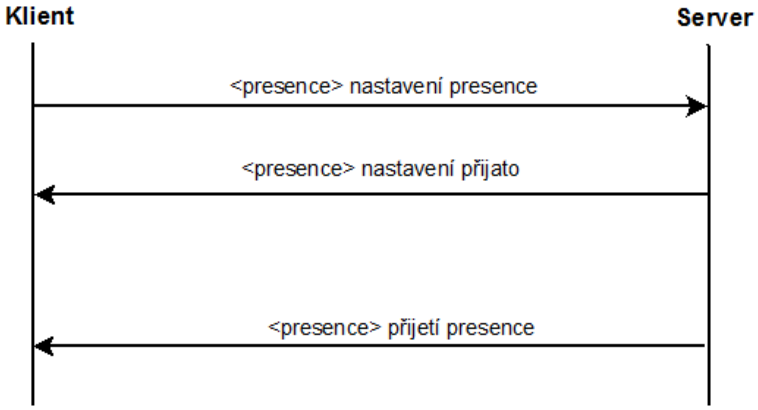
Zprávou klient žádá o seznam kontaktů. Server tedy seskládá všechny potřebné kontakty, které uživatel má a pošle je zpět:

```
<iq id='uid:4bdf5228:66334873' type='result' xmlns='jabber:client' from='parma@lizols.cz'>
        <query xmlns='jabber:iq:roster'>
                <item jid='slozil@lizols.cz' name='Zbynek Slozil' subscription='both'>
                        <group>Jabber\Borci</group>
                </item>
                <item jid='sikora@lizols.cz' name='Tomas Sikora' subscription='both'>
                        <group>Jabber\Pratele</group>
                </item>
        </query>
</iq>
```

Zpráva obsahující kontakty. Každý item je jeden kontakt uživatele, u kterého jde definovat jméno (name), které se bude klientovi zobrazovat, stupeň autorizace (subscription), kde both znamená oboustranná (každý vidí zprávu druhého), a skupinu (group).

Presence (Obr.6)

Presence poskytuje aktuální stav uživatele ze seznamu kontaktů.



Obr.6 Presence

Příklad průběhu:

Nejdříve klient žádá o nastavení své presence:

```
<presence from='parma@lizols.cz' xmlns='jabber:client'>
        <priority>0</priority>
        <show>away</show>
        <status>Jsem pryč!</status>
        <c xmlns='http://jabber.org/protocol/caps' hash='sha-1' node='http://camaya.net/gloox'
ver='MCjlOscOkUNT+8QzA5fFJj2oMSU='/>
</presence>
```

Ve zprávě je i nastavení priority, která určuje, který zdroj má být vybrán, je-li použit „bare JID“. Dále

se objevuje nastavení typu presence (show), jehož druhy nejsou pro server důležité, on je pouze rozesílá ostaním. Nakonec zpráva, která se u presence objeví (status). Server posílá pozitivní odpověď:

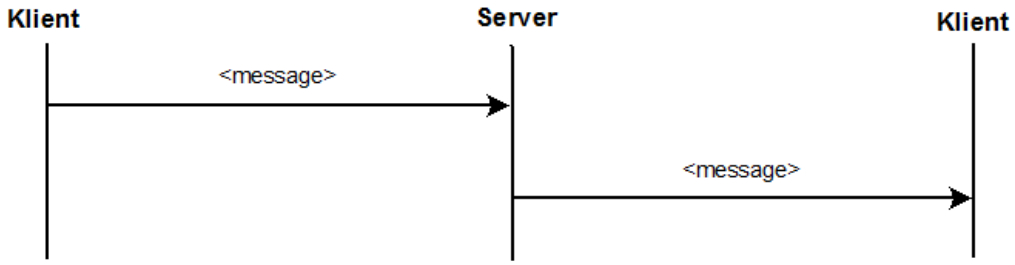
```
<presence from='parma@lizols.cz/gloox' to='parma@lizols.cz/gloox' xml:lang='cs'/>
```

Pokud je někdo ze seznamu kontaktů právě připojen, server mu pošle zprávu o změně statusu uživatele. Také pokud uživatel změní kdykoli svůj presence, bude rozeslána tato zpráva všem, kteří jsou připojeni:

```
<presence from='parma@lizols.cz/gloox' to='sikora@lizols.cz/gloox' xml:lang='cs'>
        <priority>0</priority>
        <show>away</show>
        <status>Jsem pryč!</status>
</presence>
```

Zpráva (Obr.7)

Posledním důležitým tagem je message, sloužící k posílání zpráv.



Obr.7 Zpráva

Příklad průběhu:

Klient pošle:

```
<message to='parma@lizols.cz' id='uid:4bdf5242:19495cff' type='chat' from='sikora@lizols.cz'
xmlns='jabber:client'>
        <body>hello</body>
        <thread>glooxuid:4bdf5242:74b0dc51</thread>
        <x xmlns='jabber:x:event'>
                <offline/>
                <delivered/>
                <displayed/>
                <composing/>
        </x>
        <active xmlns='http://jabber.org/protocol/chatstates'/>
</message>
```

Server přepošle tuto zprávu danému existujícímu uživateli.

Jingle

Je rozšíření, která nabízí možnost vymezení pravidel pro navázání binárního spojení. Rozšíření jsme chtěli po dohodě se Zdeňkem Parmou použít pro upřesnění pravidel navázání multimediálního spojení. XMPP jak už bylo řečeno jsou tvořeny XML soubory posílanými mezi klienty a servery. Nevýhoda v tomto konceptu je nepřítomnost binárního spojení, které vyžaduje jak audio nebo video přenos. Právě Jingle obchází tento nedostatek. Pomocí tohoto rozšíření protokolu se server a klient dohodnou o způsobu navázání.

Po úpravách tohoto protokolu ho lze použít s mnoha multimediálními protokoly pro přenos audia/videa. Příkladem může být RTP[9], který je určen k tomuto účelu.

3.1.4. Knihovny pro implementaci XMPP

Existuje velké množství knihoven pro implementaci XMPP, ale většinou se zaměřují pouze na klienty. Popíšu pouze dvě knihovny, ze kterých jsem vybíral a zaměřím se hlavně na možností parserování XML zpráv.

Knihovna gloox [8]

Knihovna založená na jazyce C++. Má obsáhlou dokumentaci a je vyvíjena pod licenci GNU GPL, proto zkoumání kódu nebo úprava samotné knihovny není problém. Také je velmi multiplatformní, zvládá systémy jako Windows, Linux, Mac OS X a další. Knihovna je zaměřena na implementaci klienta, a má mnohé funkce, které řeší konektivitu a tvorbu XML zpráv bez účastí vývojáře. Pro mě je však důležitá část parserování a paměťového řízení. Knihovna tvoří velké třídy pro ukládání přijatých XML zpráv, ze kterých je dostatečný přístup ke všem elementům. Na mě je ale způsob použití v této knihovně zbytečně robustní a složitý.

Knihovna iksemel [9]

Velmi malá knihovna (zkompilovaná zabírá kolem 30kB) implementovaná v jazyce C na licenci GNU LGPL (možnost komerčního použití). Oficiálně pracuje v POSIX[10] prostředí (použití například s kompilátorem gcc) a i v prostředí Microsoft Windows (pomocí kompilátoru MINGW, založeného na gcc, ale pro MS Windows). Tak jako gloox i tato knihovna je určena pro implementaci klienta, má ale mnohem méně dovednosti. Knihovna zvládá automaticky posílat pouze protokol XMPP Core, ostatní části si musí vývojář vytvořit sám. Proto se důležitou části stal DOM a SAX parserer, které jsou v knihovně. Oba v moji implementaci serveru využívám. Při tvorbě stromu pro DOM parserer se knihovna snaží o co nejmenší přenosy v paměti a o jednoduchost použití. Také podporuje šifrování TLS[11] a SASL[12], které jsou v XMPP podporovány.

Výhody knihovny:

- Vysoká přenositelnost, a možnost kompilace na POSIX systémech i MS Windows.

- Výhodný pro přenosné systémy. Kód je modulární a je možnost oddělit nepotřebné části.

- Nízká paměťová náročnost a rychlé algoritmy pro parserování.

- Poskytuje SAX, DOM a XMPP parser.

- Pro zabezpečení XMPP je podpora TLS (pomocí gnutls) a SASL.

- Běží pouze na UTF-8 (jiné kódování není u XMPP ani povoleno).

- Dobrá dokumentace s příklady použití.

3.2. Knihovna Pthread

Knihovna API řešící tvorbu a manipulaci s vlákny na úrovní operačního systému dle standardu POSIX. Je implementovaná v jazyce C na všech UNIX systémech. Tato knihovna rozšiřuje

9 Real Time Protocol, protokol pro posílání audio a video dat po internetu.
10 Portable Operating System Interface, přenositelné rozhraní mezi operačními systémy.
11 Transport Layer Security, kryptografický protokol pro zabezpečenou komunikaci.
12 Simple Authentication and Security Layer, ověřovací metoda mezi klientem a serverem.

paralelismus programů a má mnoho výhod i nevýhod oproti tvorbě více procesů.

Výhody:
- sdílený paměťový prostor
- rychlé přepínání mezi vlákny
- menší spotřeba paměti
- mnohdy rychlejší tvorba a rušení vláken

Nevýhody:
- pokud se zhroutí jedno vlákno, spadne celá aplikace
- stejná oprávnění pro všechna vlákna
- sdílení mnoha signálů
- sdílené file descriptory (sockety, otevřené soubory, …)

Vlákna přistupuji ke sdílené paměti pomoci globálních proměnných (proměnná zapsána v kódu mimo prostor funkce) nebo jako v mém případě mohu pří volání vlákna předat ukazatel na datovou oblast (musí se zaručit, že data budou dostupna po celou dobu běhu vlákna). Vlákna mají svůj vlastní zásobník a lokální proměnné nesdílí, proto jediný přístup ke sdíleným datům v mé implementaci je přes zmíněný ukazatel na třídu, ve kterém se řeší i kritické sekce.

Knihovna se dá rozdělit do tří částí:

Vlákna – například tvorba, volání nebo likvidace samotného vlákna
Mutexy – knihovna implementuje vlastni způsob řešení souběhu pomocí mutexů
Podmínkové proměnné – pasivní čekání vlákna na splnění nějaké vnější podmínky

Lze vidět že vlákna například neimplementuje semafory, ten lze však přidat pomoci hlavičky semaphore.h a použít podobně jako při práci s více procesy (semafory nejsou zatím oficiálně v POSIX standardu pro vlákna). Semafory nejsou důležité jenom proto, že mohou udávat svojí velikost, ale také proto, že při opuštění čekání se vlákna probouzí ve stejném pořadí jako usínaly na rozdíl od mutexu, který je pouští náhodně.

3.3. Možnosti konektivity v Linuxu

Nyní budu popisovat TCP spojení, ale UDP je velmi podobné. Konektivita na sítí se provádí pomocí socketů. Na začátku se musí definovat naslouchající socket, který se také nastaví. Po funkci listen začne hlavní socket přijímat nová spojení. Ty příjme funkcí accept a uloží nový socket pro zpracování příchozích zpráv.

Nastavení socketu je mnoho, stačí přečíst manuálové stránky, pro mě je je však důležitá hlavně část s nastavením blokového a neblokového režimu. Neblokový režim se od blokového liší zásadně tím, že při volání čtení ze socketu se funkce nezablokuje při nedostupnosti zprávy, ale vrátí chybovou hodnotu. Tento způsob se používá při čtení (zápisu) v množině socketů. Na rozdíl blokový režim by musel mít pro sebe na každý socket svůj vlastní proces (vlákno), tento koncept jsem už ale v úvodu zavrhl.

Dále se tedy budu zabývat pouze neblokovému režimu, který se taky nejčastěji používá. Důležitou částí se tedy stává funkce, která bude kontrolovat zda na nějakém ze skupiny socketů nenastal požadovaný stav. Pokud se tak stalo, musí se proces (vlákno) probudit a daný socket zpracovat. Tyto hlídající funkce dokáží kontrolovat, všechny stavy, které se na socketu mohou objevit, to znamená, že jde například řešit v jednom procesu (vlákně) naráz zápis i čtení dat. Velmi to snižuje zátěž celého systému, protože operace se síťovou kartou jsou pro procesor velmi pomalé a než se příjme další síťový paket může procesor obsloužit přijatá data (nic se však nemá přehánět, hlavně když se v obslužné části objeví další I/O operace, například zápis na disk).

Seznam funkcí pro tvorbu spojení v mé implementaci:
socket – vytvoří hlavní socket pro naslouchání

setsockopt, fcntl – slouží k nastavení socketu
bind – spojí IP adresu a port se socketem
listen – nastaví maximum spojení a zapne samotné naslouchání
accept – příjme z hlavního socketu nové spojení a vrátí nový klientský socket
recv, send – posílá a přejímá zprávy, u mě pracují v neblokovaném režimu

Chybí zde funkce pro čekání v množině socketů. Tuto část popíšu podrobněji, protože existuje velké množství těchto funkcí a chci poukázat proč jsem vybral tu, kterou jsem použil v mé implementaci.

Existují zásadně dva druhy, funkce select a funkce poll. Funkce select se už zásadně nepoužívá a ani se nedoporučuje, jeho implementace už je podle vývojářů zastaralá a nahrazuje ho už zmíněný poll. Oba, ale dělají totéž, ale trochu jiným způsobem. Select musí množinu socketů pokaždé vytvořit před spuštěním samotné funkce a to pomocí definovaných maker. To by se mohlo zdát pro větší množství socketů docela nevýhodné. Za to poll, vytvoří předem množinu struktur, ve které jsou i dané sockety a po projití samotné funkce poll se v každé struktuře změní stavová proměnná, která definuje případnou změnu. Je to vlastně podobný způsob jen se neprochází celá množina při tvorbě, ale až při zjišťování zmíněného stavu.

Obě tyto metody se tedy dají lehce použit podle toho jaké má vývojář požadavky, ovšem u mě byla podmínka, že naslouchání, čtení a zápis dat musí být ve zvláštních vláknech. Zde ihned nastává problém, když funkce accept přijme nové spojení a vloží je k ostatním, tak čekající funkce select ani poll nepoznají, že se tak stalo. Obě funkce vlastně čekají na množinu, ve které nové spojení není. Až by například byly vyrušeny nějakou událostí z jiného socketu, například přijatou zprávou, množina by se znova načetla a problém by zmizel (i když poll načítá množinu jen jednou, tak existuje možnost novou dodatečně přidat). První řešení co mě napadlo bylo pomocí signálů. Obě funkce select i poll mají varianty pselect a ppoll, které hlídají i událost na definovaném signálu. Pokud se signál objeví, funkce je probuzena a vrátí dohodnutou návratovou hodnotu. To by znamenalo, že po přijetí nového spojení pomocí accept, bych probudil pomocí signálu všechny pselecty a ppoly, které by aktualizovaly své množiny. Toto řešení sic se zdá jednoduché je pro implementaci komplikované (i když zajímavé). Po pár testech tohoto řešení jsem zkusil najít jiné.

Nové řešení jsem našel ve zvláštní verzi funkce poll, přesněji epoll. Podle testů z vývojových stránek má dokonce nejlepší rychlostní výsledky a to díky své architektuře.

Epoll se skládá ze tří funkcí:
epoll_create – vytvoří množinu socketů o předpokládané velikosti
epoll_ctl – pomocí této funkce se vkládají, odebírají a přenastavují sockety a události
epoll_wait – hlavní funkce, která se zablokuje, dokud se neobjeví požadovaná událost

Rozdíl oproti ostatním funkcím je ta, že pokud přidám nový socket do množiny, funkce epoll_wait na ní reaguje, i když je v ní zrovna zablokované vlákno. Přestože tato vlastnost není nikde popsaná, otestoval jsem ji a nenašel jsem v ní žádné problémy. Samotný návrh funkce přidává taktéž na výkonu tím, že je množina po celou dobu celistvá a ne jak po funkci select, kde v ní zůstanou jen ty sockety, na kterých se stala událost. Funkce epoll_wait vrací novou množinu, kopii té originální, která obsahuje jen ty sockety, na kterých se objevila definovaná událost. Pokud tedy existuje obrovská množina socketů, epoll jí nemusí pokaždé procházet.

Proto zásadně v implementací využívám funkcí epoll. Jen při zápisu mám obyčejný poll, protože v množině socketů je jen jeden záznam, ve kterém se socket mění dle odesílané zprávy. Změny socketů jdou lépe provádět se strukturou než s pomocí funkce epoll_ctl. Tento poll však plní pouze kontrolní funkci, hlídající zda socket je stále dostupný.

4. Implementace

Základní koncept i použité technologie byly vysvětleny, nyní uvedu bližší řešení samotné implementace.

4.1. Zkompilování a testování

Protože očekávám, že zdrojový kód se bude kompilovat na danou architekturu, jsou dodány zdrojové soubory, které jsou v adresáři *src*. V hlavním adresáři je přidán *makefile*, a pro zkompilování stačí spustit program make. Podmínkou je mít nainstalovaný samotný program make tak i kompilační program g++. Po zkompilování se vytvoří spustitelný soubor *xmppserver* v hlavním adresáři.

Pro spuštění stačí zapnout vytvořený binární soubor. Problémem jsou pouze uložení uživatelé na serveru, ti jsou totiž přidání přímo v kódu a nepoužívá se žádná databáze ani soubor pro načtení. Pro první verzi tohoto programu to nebylo prioritou a není problém pro testovací účely upravit hodnoty ve zdrojových kódech. Přidání nebo změna se provádí v souboru main.cpp dle už existujících záznamu.

Hned po zapnutí programu aplikace běží a vypisuje zprávy které přes ní procházejí. Výpis je zde jen pro testovací účely a ponechávám ho pro názornost.

Pro vypnutí aplikace stačí stisknu kombinaci CTRL-C, aplikace signál odchytí a správně ukončí aplikaci.

4.2. Obecný popis běhu programu

Po té co je aplikace spuštěna jsou uživatelé načtení do příslušné třídy User. Poté jsou zavolány všechny vlákna - naslouchání, čtení a zápis do socketu a samotné jádro. Naslouchací vlákno nastaví nezbytné nastavení pro naslouchání na portu (hodnoty načte ze třídy General) a čeká na spojení klienta. Ostatní vlákna taktéž čekají na vstupní podnět pro probuzení.

Po přijetí spojení je naslouchací vlákno probuzeno a přijatý socket je uložen do třídy Resource. Vlákno poté znova usne a čeká na další klientské spojení.

Po přijatém spojení je nyní možné přijímat zprávy. Čtecí vlákno na ně čeká na všech socketech, které přijalo naslouchací vlákno a jsou uložené ve tříděn Resource. Pokud je detekována přijatá zpráva, vlákno se probere, vytvoří místo o dané velikosti v paměti a do něj zkopíruje přijatou zprávu. Poté uloží ukazatel na tuto zprávu do čtecího bufferu. Vlákno se pak znova uspí dokud nedetekuje novou příchozí zprávu.

Vlákno jádra čeká na jakékoli změny ve čtecím bufferu, proto poté co se v něm objeví ukazatel na příchozí zprávu se probudí. Zpráva je rozkouskována knihovnou iksemel a popřípadě zavolána daná funkce pro vyřešení daného typu tagu. Příchozí zpráva může být v tomto vláknu vymazána z paměti a místo ní vytvořená jiná zpráva nebo může by upravena a poslána dál. Odpovědi na zprávu jsou zapisovány do zapisovacího bufferu.

Zapisovací vlákno vyčkává na změny v zapisovacím bufferu, ze kterého vlákno vyjme ukazatel a socket, na který má být zpráva poslána. Poté co se tak stane je dané místo v paměti, které zpráva zaplňovala, uvolněno.

Celý tento koloběh se stále opakuje, s tím že zpráv může za sebou chodit obrovské množství a pro vyrovnání rychlosti vláken slouží už zmíněné buffery.

4.3. Rozdělení aplikace

4.3.1. Podle souborového systému

Jako výstup po kompilaci je jen jeden spustitelný soubor programu, ale ovšem souborů pro kompilaci je víc (k cpp patří i hlavičky kromě main.cpp).

main.cpp – obsahuje pouze volání vláken a naplňuje program daty (uživatelé)

connection.cpp – obsahuje vše ohledně spojení. Jsou zde tři funkce a každá z nich je spuštěna jako samostatné vlákno z main.cpp

core.cpp – obsahuje zpracování příchozích zpráv a tvorby nových pro připojeného klienta

data.cpp – datová část aplikace, jsou zde všechny funkce, které řeší přístup k datům. Řeší i kritické sekce paralelního přístupu. Jsou v něm také buffery které řeší komunikaci mezi vlákny.

4.3.2. Podle vláken a jejich funkcí

Jak už jsem psal v konceptu, aplikace je rozdělena na čtyři hlavní části. Každá část je tvořena spuštěným vláknem z knihovny pthread, které většinou čeká na nějaký vnější podmět.

Vlákno pro naslouchání

Jeho práce je naslouchat na daném portu (pro xmpp je to obecně 5222). Vlákno nejdříve nastaví nezbytné parametry pro naslouchání a pak najede do cyklu. V cyklu poté čeká na žádosti o spojení, které, pokud je přijato, se přidá k ostatním na dané datové místo. Toto vlákno není nijak zatěžováno, proto se v něm bude uskutečňovat u naslouchání pro příchozí multimediální spojení.

Vlákno pro čtení

Práce tohoto vlákna je čekat na příchozí zprávy ze všech socketů které přijalo naslouchací vlákno. I toto vlákno je zaseknuté v cyklu a čeká dokud se na některém socketu neobjeví zpráva. Pokud se tak stane, pro zprávu je vytvořeno místo v paměti a je odesláno do vstupního bufferu. Do bufferu je také přidán záznam, ze kterého socketu zpráva přišla, aby jádro programu zprávy od sebe rozeznalo.

Vlákno pro zápis

Na rozdíl od čtecího vlákna, zápisové vlákno čeká na změnu ve výstupním bufferu (čeká na semaforu). Pokud se v něm objeví nějaký ukazatel na zprávu, je zkontrolován zda je daný socket volný a poté je samotná zpráva odeslána danému klientovi. Zpráva je pak smazána z paměti, aby uvolnila místo ostatním.

Vlákno jádra programu

Toto je nejdůležitější vlákno. Stará se zpracování zpráv, které se objeví na vstupním bufferu. Rozkóduje příchozí zprávu a výsledek zapíše do výstupního bufferu. Počet zpráv, které je potřeba na vstupu nebo kolik jádro vytvoří při jednom průchodu může být různý. Například při úvodní žádosti o spojení je na vstupním bufferu jedna zpráva a po zpracováni jsou na výstupu dvě zprávy, které jsou poté zvlášť poslány. Buffer zajistí že jsou rychlostní rozdíly ošetřeny, proto i kdyby jádro vytvořilo dalších sto nových zpráv, tak jedině co se stane je, že se zaplní buffer.

Po bližším zkoumání a troše úprav by se například dal zvýšit počet vláken pro jádro programu, které by mohly jet paralelně a vytížit lépe více-jádrový procesor, na kterém by běžel. Tato koncepce je ovšem složitější, protože některé XML přicházejí v daném pořadí a musí se tak i vyřešit.

Hlavní program.

Neděla nic než pro testovací účely zaplní program daty a poté spustí ve správné pořadí všechny nebytné vlákna.

4.3.3. Datové rozdělení

Všechna data jsou uloženy v jedné velké třídě. Odkaz na tuto třídu je předáván každému vláknu, aby měl přístup ke všem potřebným proměnným, které jsou pro jeho chod důležité. Třída zatím obsahuje dalších pět podtříd, které slouží k různým účelům.

Podtřída General

Tato třída se stará o uložení globálních proměnných, které mají obecný význam. Také předpokládám že mnoho z těchto proměnných bude v budoucnu načteno z konfiguračního souboru, ve kterém bude moc uživatel měnit nejdůležitější nastavení.

Podtřída Resource

Třída k uložení daných spojení a jeho vlastností. Po té co je přijato jakékoli spojení, je uloženo zde a dalšími příchozími zprávami od klienta jsou jeho vlastnosti upřesněny (název, priorita). Jádro této třídy je tvořeno vektorem, který se stal kritickou sekcí.

Podtřída User

Je velmi podobná třídě Resource. Na rozdíl od ní se však nestará o spojení, ale o uživatelé, kteří tyto spojení „vlastní“. Tak jako Resource se stal kritickou sekci pro paralelizaci vektor, který nastavení každého uživatele ukládá.

Podtřída WriteBuffer

Vstupní buffer, který řeší předávání dat mezi zapisovacím vláknem a vláknem jádra.

Podtřída ReadBuffer

Velmi podobný jako WriteBuffer, ale řeší předávání mezi čtecím vláknem a vláknem jádra.

4.4 Bližší zkoumání funkcí

Popíšu jen funkce které jsou zajímavé nebo složitější.

4.4.1. Funkce pro naslouchání

Slouží k příjmu příchozích TCP spojení od klienta. Zde bude i implementované příchozí multimediální spojení.

```
25 void * threadListenTCP(void * arg)
26 {
27    // vytvoreni spojeni s daty
28    Data * data = (Data *) arg;
.
.
53    // nastaveni do neblokovaciho rezimu
54    int old_flag = fcntl(lis_socket,F_GETFL,0);
55    if (fcntl(lis_socket,F_SETFL, old_flag | O_NONBLOCK) == -1) {
.
.
```

```
75   // vytvoreni poll struktury pro naslouchani
76   int lis_epoll = epoll_create(1);
77   epoll_event lis_events;
78   lis_events.data.fd = lis_socket;
79   lis_events.events = POLLIN;
80   epoll_ctl(lis_epoll, EPOLL_CTL_ADD, lis_socket, &lis_events);
81   epoll_event event;
82   // nyni naslouchа na portu
83   int cli_socket;
84   while (true) {
85     if (epoll_wait(lis_epoll, &event, 1, -1) == -1) {
86       cerr << "Error: listen poll" << endl << flush;
87       return NULL;
88     }
89     // vrati novy socket
90     if ((cli_socket = accept(lis_socket, NULL, NULL)) == -1) {
91       cerr << "Error: listen accept" << endl << flush;
92       continue;
93     }
94     else {
95       data->resource->add(cli_socket);
96     }
97   }
.
.
102 }
```

Je dobré se zmínit o předávání dat přes parametry vlákna (řádek 28). Tohle je jediný způsob, jak se funkce dostane ke globálním proměnným. Celý server běží v neblokovaném režimu (řádek 55), proto se zde musí použít nějaký select nebo poll. V mém případě je to speciální epoll, který je rozdělený na tři funkce, vytvoření (řádek 76), nastavení (řádek 80) a čekání na událost (řádek 85). Důležitá je potom část přidání nového socketu k ostatním (řádek 95).

4.4.2. Funkce pro čtení zpráv

Jediným cílem této funkce je přečíst příchozí zprávy.

```
105 void * threadReadTCP(void * arg)
106 {
.
.
113   // nyni ceka na zprávy
114   epoll_event r_events[max_connection];
115   int count_events;
116   int lenght_message;
117   char * message;
118   while (true) {
119     count_events = epoll_wait(read_epoll, r_events, max_connection, -1);
120     if (count_events == - 1) {
121       cerr << "Error: cteni poll" << endl << flush;
122       break;
```

```
123       }
124       else {
125         for (int i = 0; i < count_events; i++) {
126           message = new char[max_message];
127           lenght_message = recv(r_events[i].data.fd,(void *) message, (max_message - 1), 0);
128           if (lenght_message == 0) {
129             cout << "Socket " << r_events[i].data.fd << " se odpojil na tvrdo." <<  endl <<
flush;
130             delete[] message;
131             data->resource->erase(r_events[i].data.fd);
132             break;
133           }
134           message[lenght_message] = '\0';     // pridani nuloveho znaku
135
136           data->read_buffer->push(message, r_events[i].data.fd);
137         }
138       }
139     }
140    return NULL;
141 }
```

Tak jako v naslouchací funkci i zde se používá množina socketů, ale namísto aby si ji funkce vytvářela sama, přejímá se už připravená z globálních dat. Poté co se objeví nějaká zpráva na socketu (řádek 119), přečte se(řádek 127) a přidá se k ní ukončovací znak pro zjištění délky řetězce řadě funkcí. Zpráva je uložena do paměti a poté její ukazatel se socketem zapsán do čtecího bufferu (řádek 136).

4.4.3. Funkce pro zápis zpráv

Funkce trochu podobná funkci čtení zpráv.

```
143 void * threadWriteTCP(void * arg)
144 {
.
.
151    pollfd write_poll;
152    write_poll.events = POLLOUT;
153    write_poll.revents = 0;
154    char * message;
155    bool last_message;
156    while (true) {
157      message = data->write_buffer->pop(&write_poll.fd, &last_message);
158           cout << "zapisuje: " << message << " na " << write_poll.fd << endl << flush;
159      while (poll(&write_poll, 1, -1) == -1);
160      else {
161        if ((write_poll.revents & POLLOUT) != 0) {
162          send(write_poll.fd,(void *) message, strlen(message), 0);
163          delete[] message;
164          if (last_message == true) {
165            data->resource->erase(write_poll.fd);
166          }
```

```
167         }
168         write_poll.revents = 0;
169       }
170     }
171     return NULL;
172 }
```

Oproti předchozím dvou funkcí, tato čeká na příchozí ukazatel ze zapisovacího bufferu (řádek 157). Data odebírá po jednom a pomocí poll kontroluje zda je socket pro zápis připraven (řádek 159). Pokud není, zkusit to znova. Nepřipravenost socketu je ovšem extrémní situace, protože se zapisuje přes systémové buffery, které vytíženost síťové karty ošetří. Nezbytným úkolem po poslání zprávy (řádek 165) je její vymazání, jinak by server mohl paměťově narůstat. Další důležitou části je také řádek 167 až 169, který vymaže socket z globální části po poslání ukončovací zprávy.

4.4.4. Hlavní funkce jádra

Jedná z nejdůležitějších částí serveru. Tato hlavní funkce vlákna provádí přerozdělování tagů XML a volání příslušných funkcí pro vyřešení.

```
61 void * threadCore(void * arg)
62 {
.
.
77    while (true) {
78      receive_message = data->read_buffer->pop(&receive_socket);          // ceka na buffer
79      cout << "cte: " << receive_message << endl << flush;
80
81      test_parser = iks_sax_new((void *) data, pr_tag, NULL);   // sax pro testovani
82      type_message = iks_parse (test_parser, receive_message, 0, 1);
83
84      if (type_message == IKS_IQ) {
85        // ********************** IQ
86        if (typeIq(data, receive_socket, receive_message) == false) {
87          typeEndStream(data, receive_socket);
88        }
89      }
90      else if (type_message == IKS_PRESENCE) {
91        // ********************** PRESENCE
92        if (typePresence(data, receive_socket, receive_message) == false) {
93          typeEndStream(data, receive_socket);
94        }
95      }
96      else if (type_message == IKS_MESSAGE) {
97        // ********************** MESSAGE
98        if (typeMessage(data, receive_socket, receive_message) == false) {
99          typeEndStream(data, receive_socket);
98        if (typeMessage(data, receive_socket, receive_message) == false) {
99          typeEndStream(data, receive_socket);
100       }
101     }
102     else if (type_message == IKS_STREAM_START) {
```

```
103         // ********************** STREAM start
104         delete[] receive_message;
105         typeStartStream(data, receive_socket);
106       }
107       else if (type_message == IKS_STREAM_STOP) {
108         // ********************** STREAM konec
109         delete[] receive_message;
110         typeEndStream(data, receive_socket);
111       }
.
.
133       iks_parser_delete(test_parser);
134     }
135     return NULL;
136 }
```

Po vstupu co cyklu se funkce zastaví (řádek 78) a čeká na nové zprávy z čtecího bufferu. Poté co je zpráva přijata, je použit SAX parser (řádek 81, 82) pro učení základního tagu zprávy. Poté je zavolána daná funkce dle typu pro vyřešení.

4.4.5. Ostatní funkce jádra

Ostatní funkce se většinou řeší zpracování příchozí zprávy a tvorby odpovědi (řešeno v části XMPP protokolu). Pro průchod i tvorbu XML dokumentu je většinou použit DOM algoritmus, který vytváří XML strom v paměti.

Funkce *pr_tag* slouží algoritmu SAX pro zjištění tagů XML. V normálním případě by SAX prohledával tagy dál, já si však definoval své vlastní návratové hodnoty, které knihovna iksemel, ve které je SAX implementován, vyhodnocuje jako chybu a parserování opouští (najde první tag a poté ukončí parserování).

4.4.6. Funkce z datové třídy

Skoro všechny funkce z této třídy nejsou nijak zajímavé, protože se starají jen o uložení a načtení dat pro běžící vlákna. Výjimkou je řešení paralelního přístupu, kterou ještě popíšu podrobněji.

4.5. Řešení kritických sekcí

Obecně jsou v aplikaci dva typy těchto kritických sekcí.

1. **Předávání dat mezi vlákny pomoci bufferů**

   Tyto buffery jsou vlastně fronty ze standardní knihovny C++ upravené pro paralelní přístup, který je řešen pomocí klasického problému „Producent konzument“.

   Jádrem algoritmu je je fronta (u mého programu už zmíněná fronta ze standardní knihovny), kterou plní producent (u mě například čtecí vlákno). Pokud by fronta byla plná, producent bude čekat dokud se neuvolní místo. Toto místo uvolňuje konzument (pokud počítám s minulým příkladem producenta jako čtecí vlákno, tak by to bylo vlákno jádra), který, pokud už nejsou data, vyčkává, až producent vloží nový prvek do fronty.

   Je mnoho řešení problému a jsou si všechny podobné.

   Producent:

```
wait(free);             // snižuje počet volných pozic
lock(mutex);            // vstup do kritické sekce
pridat_prvek();
unlock(mutex);          // výstup z kritické sekce
signal(used);           // zvyšuje počet použitých pozic
```

Konzument:

```
wait(used);             // snižuje počet použitých pozic
lock(mutex);            // vstup do kritické sekce
odebrat_prvek();
unlock(mutex);          // výstup z kritické sekce
signal(free);           // zvyšuje počet volných pozic
```

Algoritmus je jednoduchý, v mém případě jsem použil jeden mutex (z knihovny pthread) a dva semafory. Mutex pro přístup do kritické sekce a semafory pro počítání volných a použitých pozic ve frontě.

Také by problém šel vyřešit pouze pomocí semaforů, ale mutex je v linuxu o něco rychlejší, proto jsem zvolil ten. Nevýhodou je však, že mutex nespouští po čekání procesy ve stejném pořadí jak do něj vstoupily.

Velikost fronty ze standardní knihovny je teoreticky nekonečna, ale semafory ji určují hranice, které jsou dány ze třídy General.

2. Přístup ke sdíleným datům ve třídách

Tento způsob je použit k přístupu k datům tříd Resource a User. Prostor pro uložení je většinou tvořen polem nebo vektorem a nad nimi pracuje algoritmus „Čtenáři a písaři“ s prioritou písařů. Tato priorita je zvolena, protože předpokládám mnohem větší čtení dat než zapisování. Také by se data mohli stát neaktuálními, pokud by se zapisování nedostalo na řadu.

Obecným účelem tohoto problému je aby písaři (vlákna) mohli číst data současně a navzájem se neomezovali. Při zapisování se však všechny vlákna zablokují a počkají dokud se nezmění data. Toto je nejednoduší způsob jak zvýšit výkonost paralelní aplikace.

Čtenář:

```
wait(reader);               // vstup do kritické sekce uzamykání čtenáře
lock(readcountmutex);       // vstup do kritické sekce proměnné readcount
readcount++;                // zvýší počet čtenářů přistupující k datům
if (readcount == 1)
        wait(writer);       // zablokuje zápis
unlock(readcountmutex);     // výstup z kritické sekce proměnné readcount
signal(reader);             // výstup z kritické sekce uzamykání čtenáře

cteni();

lock(readcountmutex);       // vstup do kritické sekce proměnné readcount
readcount--;                // sníží počet čtenářů přistupující k datům
if (readcount == 0)         // pokud už neexistuje čtenář, pravda
        signal(writer);     // uvolní zápis
unlock(readcountmutex);     // výstup z kritické sekce proměnné readcount
```

Písař:

```
lock(writecountmutex);          // vstup do kritické sekce proměnné writecount
writecount++;                   // zvýší počet písařů přistupující k datům
if (writecount==1)
        wait(reader);           // zablokuje čtení
unlock(writecountmutex);        // výstup z kritické sekce proměnné writecount
wait(writer);                   // vstup do kritické sekce zápisu

zapis();

signal(writet);                 // výstup z kritické sekce zápisu
lock(writecountmutex);          // vstup do kritické sekce proměnné writecount
writecount--;                   // sníží počet čtenářů přistupující k datům
if (writecount==0)
        signal(reader);         // uvolní čtení
unlock(writecountmutex);        // výstup z kritické sekce proměnné writecount
```

V mém případě jsou použity dva mutexy a dva semafory. I když by se zde daly použit jenom mutexy, podle mého názoru je zde důležitější pořadí výstupu z čekání na semafor. To by mělo také zaručit už zmíněnou aktuálnost dat.

Ostatní sdílené data nejsou nijak chráněna, například proto, že jejich změna neovlivní běžící program (proměnná načtená jenom jednou) nebo riziko souběhu je velmi malé a zatím se neřešilo.

4.6. Multimédia

I když implementace tohoto úkolu nebyla uskutečněna, navrhl jsem základní koncept

Klient by pomocí Jingle zažádal server o uskutečnění multimediálního spojení. Ten by to oznámil druhému klientovi, se kterým se chtěl spojit. Parametry spojení by se taktéž nastavily pomocí Jingle. Poté by server naslouchal na portu, a čekal než by se klienti připojili pomocí UDP. Až by přicházeli data, server by je pouze přeposílal druhému klientovi.

Tento jednoduchý koncept má mnoho chyb. Například server by měl naslouchat jen na jednom portu (je zbytečné, aby pro každé pro každé spojení měly klienti jiný port) a tak se musí příchozí UDP spojení od sebe rozeznat. Přemýšlel jsem o metadatech , které by obsahovaly autentizaci připojeného uživatele. Tím bych mohl určit komu spojení patří a správně přeposílat binární data.

Pro zjednodušení se na straně klienta bude asi používat knihovna, která by posílání řešila sama (např. GStreamer[13]), proto bych se musel přizpůsobit způsobu uložení metadat v této knihovně.

Ostatní věci, jako použitý druh kodeků, by mě v prvotní fázi nezajímaly, protože server by data jenom přeposílal, ale nikam neukládal. Výjimkou by byla zvláštní komunikace po síti, kterou by kodek mohl mít, například použití více spojení naráz.

13 Framefork pro usnadnění přeposílání multimediálních dat.

5. Závěr

Práce poskytuje návrh možného řešení implementace XMPP serveru. Důležitou části je vytvoření samotné kostry programu, ze které se dá vycházet. Mnoho knihoven řeší implementaci klienta jak parserování XML zpráv, ale i samotné spojení se serverem. V mém případě bylo potřeba navrhnout a vytvořit aplikaci od samotného základu. Ale i když jsem nevyřešil všechny dané problémy na sto procent, znalosti, které jsem při tvorbě získal jsou nedocenitelné.

Aplikace také obsahuje velké množství chyb a nedodělků, protože je to pouze první verze složitého problému. Zdokonalování může zabrat ještě mnoho času. Nyní sepíšu pár známých chyb:

- Nejsou implementované chybové zprávy zpět ke klientovi.
- Samotné kontroly hlášení chyb v aplikaci nejsou řešeny správnou cestou. Měl by se vytvořit nějaký log a možné postupy řešení chybových stavů.
- Návraty z vláken nejsou řešeny. Mělo vytvořit kontrolní vlákno, které bude kontrolovat stavy ostatních a řešit jejich problémy.
- Aplikace z vysokou pravděpodobností má úniky v paměti. I když se tento problém objevuje u mnoha jiných programů a ani o tom nemusí vědět, častou kontrolou a testováním se dá počet těchto závažných chyb omezit.
- Dalším zásadním problémem je, že zpráva je jeden řádek v XML dokumentu. Řádky mohou být přerozděleny na více přijatých zpráv nebo více řádků se může spojit do jedné zprávy. Proto se asi za buffer musí vložit část kódu, která by jádru programu zprávy správně přerozdělila.
- Z minulou chybou souvisí i chyba zbytečného posílání krátkých zpráv. Zde by mohlo být spojování více zpráv do jedné pro snížení zátěže sítě.
- Dalším nedostatkem je šifrování zpráv a autentizace. Server nemá žádné bezpečné spojení s klientem, proto se tato část měla vyřešit, bude-li tento projekt pokračovat.
- Další známe chyby byly napsány už dříve.

5.1. Plány do budoucna

Rád bych se tomuto projektu dále věnoval. Důležité bude opravit nedostatky, které tato implementace má, možná i trochu předělat, ale jako základní kostra je to dostačující. Bude důležité nejen zkoumat vlastní kód, ale i implementace jiných otevřených projektů, protože problémy které řešili jejich vývojáři budu pravděpodobně řešit i já.

Rád bych také dokončil multimediální část, která je dnes tak důležitou části IM klientů. Jako výhodu svého serveru bych chtěl zvýšit celkový výkon aplikace a aby zabírala nejméně diskového prostoru. Přemýšlel jsem i o kompilaci na obyčejných domácích routerech s nainstalovaným Linuxem, ale to už vyžaduje vysokou odladěnost celé implementace.

Literatura

[1] XMPP Standards Foundation. [online] URL: <http://xmpp.org/> [Cit. 2010-05-05].

[2] Parma Zdeněk. Jádro klienta XMPP s podporou multimédií, Ostrava. 37 s. Bakalářská práce na Fakultě elektrotechniky a informatiky Technické univerzity Ostravy. Vedoucí bakalářské práce Ing. David Seidl

[3] Sikora Tomáš. Textové a grafické rozhraní klienta protokolu XMPP s podporou multimédií, Ostrava. 36 s. Bakalářská práce na Fakultě elektrotechniky a informatiky Technické univerzity Ostravy. Vedoucí bakalářské práce Ing. David Seidl

[4] Dana Moore, William Wright. Jabber: Developer's Handbook. První vyd. Indianapolis: Developer's Library. 2004. ISBN : 0-672-32536-5.

[5] SAX [online] URL: <http://www.saxproject.org/> [Cit. 2010-05-05].

[6] Extensible Messaging and Presence Protocol (XMPP): Core [online] URL: <http://www.ietf.org/rfc/rfc3920.txt> [Cit. 2010-05-05].

[7] Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence [online] URL: <http://www.ietf.org/rfc/rfc3921.txt> [Cit 2010-05-05].

[8] Gloox [online] URL: <http://camaya.net/gloox/> [Cit. 2010-05-05].

[9] Iksemel [online] URL: <http://code.google.com/p/iksemel/> [Cit. 2010-05-05].